The Rainbow-II Gigabit Optical Network

Eric Hall, Jeff Kravitz, Rajiv Ramaswami, *Senior Member, IEEE*,
Marty Halvorson, Stephen Tenbrink, and Richard Thomsen

***Abstract*— This paper describes the Rainbow-II optical metropolitan area network (MAN), which supports 32 nodes each at 1 Gbit/s over a distance of 10–20 km. Rainbow-II uses optical wavelength-division multiplexing (WDM), in a broadcast star architecture. Each node uses a separate fixed wavelength for transmitting data and a tunable receiver for receiving one of several data streams. The network is implemented in the form of optical network nodes, each attached to a host computer via the high-performance parallel interface (HIPPI). Each network node contains protocol processing hardware to offload the protocol processing work from the host computer onto the node. The goal is to provide full gigabit-per-second bandwidth to end-user supercomputer applications. Preliminary protocol performance measurements in a testbed network are given.**

I. Introduction

OPTICAL networking using wavelength-division multiplexing (WDM) offers the potential for building large networks supporting hundreds of nodes and offering each node capacities of up to 1 Gbit/s [1]–[3]. For an earlier WDM network project, we prototyped a 32-node, 300 Mbit/s per node, WDM metropolitan-area network (MAN) called Rainbow-I [4]. This paper descibes the Rainbow-II network, which is a follow up to Rainbow-I. Rainbow-I was built primarily to gain experience with using optical tunable components and other technology in a field environment. In contrast to Rainbow-I, Rainbow-II was built to study and gain experience with higher-layer protocols and end-user supercomputer applications. The network was designed and built by IBM Research. It is currently deployed as an experimental testbed at the Los Alamos National Laboratory (LANL). LANL is making performance measurements and experimenting with gigabit applications, preliminary results of which are presented in this paper.

A. Rainbow-I Architecture

In Rainbow-I, each station has an optical transmitter that emits light at a wavelength different from other transmitters in the network. A fiber leads from each station to the network hub, which is an all-glass, totally passive star coupler with N inputs and N outputs, N being the number of stations in the network. The star coupler combines the transmissions from different stations, and at each output, we obtain approximately $(1/N)$th of the optical power from each transmitter. From the star coupler, a fiber leads to each station. Thus, the network is broadcast in nature; all transmissions reach all the stations. At each station, a tunable optical receiver selects one of the N wavelengths. Each station in the Rainbow-I network consisted of an IBM PS/2 host with an attached MicroChannel card, much like that of a token ring or Ethernet card, that provided the optical network attachment and support for implementing a simple circuit-switched media-access protocol. However, the MicroChannel was used only for control purposes; actual data entered and exited the Rainbow-I card directly using a separate serial input and output.

Manuscript received April 3, 1995; revised August 11, 1995. This work was supported in part by ARPA under Grant MDA 972-92-C-0075 and by CRADA from the Department of Energy under Grant LC9210052.

E. Hall, J. Kravitz, and R. Ramaswami are with the IBM T. J. Watson Research Center, Hawthorne, NY 10532 USA.

M. Halvorson, S. Tenbrink, and R. Thomsen are with the Los Alamos National Laboratory, Los Alamos, NM 87545 USA.

Publisher Item Identifier S 0733-8716(96)03669-4.

B. Goals of Rainbow-II

The specific goals of Rainbow-II are the following: 1) to provide connectivity to host computers using standard interfaces, 2) to deliver 1 Gbit/s throughput all the way to the application layer, and 3) to study real applications requiring Gbit/s capacities.

In order to make networks like Rainbow more useful, we must provide suitable attachment to hosts. The host interface of choice was the high-performance parallel interface (HIPPI) [5]. HIPPI is an ANSI standard for connection of data processing equipment using multiple twisted-pair copper cabling, and supports data transfer rates of 800 megabits/second. It is widely used for the interconnection of supercomputers, many of which have HIPPI interface hardware available.

Two of the major limitations of HIPPI are that it uses a cumbersome connector and cable (100 wires per connector, two connectors/cables per full-duplex connection), and that the maximum distance between HIPPI nodes is 25 m (limited by the electrical characteristics of the cable). These problems have been addressed by a version of HIPPI known as Serial HIPPI which uses HIPPI protocols and signaling methods over optical fiber. Another ANSI standard interface is the serial 1 Gbit/s fiber channel standard (FCS) [6]. The Rainbow-II network supports standard HIPPI currently. We may add a serial HIPPI or FCS interface in the future when these standards are widely implemented.

Providing gigabit-per-second speeds to each node at the physical layer is only a first step. Present-day protocols are not designed and implemented to allow applications at a node to realize gigabit-per-second throughputs from gigabit-per-second-per-node networks. Classic protocols such as TCP/IP (transmission control protocol/internet protocol) were designed to minimize transmission bandwidth requirements and not CPU overheads. The maximum packet sizes supported were small because of network-imposed constraints on error rates. These protocols were not designed to be implemented in an efficient manner and are not very well suited for implementation on multiprocessors. There are several proposed approaches to

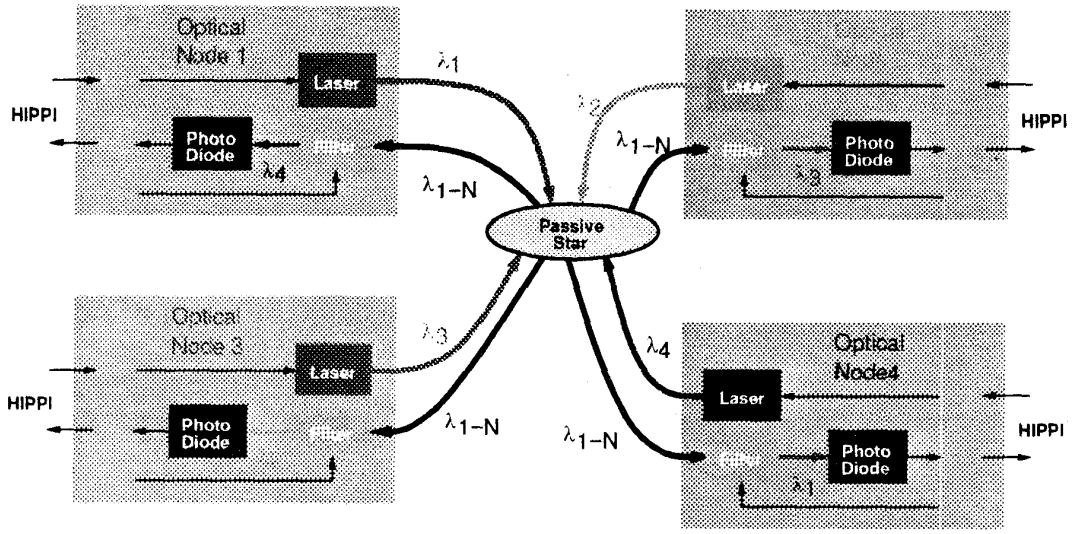

Fig. 1. The Rainbow-II network.

solving this problem. One approach is to design lightweight transport protocols that get rid of many of the inefficiencies in TCP/IP such as unaligned header fields, complicated checksum calculation, searching to locate the transmission control block for the connection each time a packet is received, etc. [7]. It has also been claimed however, that careful implementation of TCP/IP can significantly reduce the penalty in performance [8]. However, both these approaches do not solve the major problem that many hosts, particularly large parallel computers are not suited for protocol implementation, and even in those that are, e.g., a Cray, a significant portion of the CPU time is spent on protocol processing. Thus our solution is to offload all the protocol processing work from the host onto a dedicated protocol processor (PP) [9]–[12]. The protocols are implemented on on-board microprocessors with suitable hardware assists, rather than dedicated VLSI hardware, since that will give us the flexibility to experiment with a variety of protocols. We have implemented a lightweight transport protocol called optical transport protocol (OTP), to be described later.

C. Rainbow-II Implementation

Rainbow-II uses the same broadcast-star architecture as Rainbow-I. It is implemented in the form of optical network nodes (ONN's) connected to a passive star coupler as shown in Fig. 1. The ONN's are attached to host computers via HIPPI. Section II describes the hardware design of an ONN and Section II-A focuses on the protocol offload hardware. Section III gives an overview of the software, including the OTP. Section III-A describes a slim protocol developed to offload the protocol processing work from the host computer onto the ONN. Section IV gives the current status of the network and summarizes the results to date.

II. Hardware Design

The rationale above led us to design and build the Rainbow-II ONN shown in Fig. 2. The ONN consists of three logical sections: a) a host-attach section which is essentially a HIPPI interface, b) a network-attach section which contains the optical transmitter, receiver, and control hardware for the optical WDM network, and c) a protocol offload section in the middle for implementing all the protocol layers up to the transport layer.

The ONN is realized in the form of five printed circuit boards, of four types. There are two protocol processor (PP) boards, one for each direction of data travel. In addition, there is a HIPPI I/O adaptor board, and a set of two optical network attach I/O boards, one containing the optical transmitter and the receiver, and the other containing the logic to serialize/deserialize the data stream and implement the same simple media-access protocol used in Rainbow-I [4]. These boards plug into a custom backplane containing point-to-point connections for the interfaces between the boards (i.e., no shared bus). Both protocol processor boards are identical in design, with only the backplane circuitry determining which board is connected to which I/O adapter board. The interface between the PP boards and the I/O adapter boards is such that changes need not be made to the PP boards if new I/O adapter boards are designed later. In fact, it is assumed that new I/O adapter boards will be designed in future, to support enhancements to the Rainbow network (e.g., fast tuning for packet switching) or additional host attach methods (e.g., serial HIPPI, FCS, host channel).

The boards are mounted in a small, desktop or rack-mountable cabinet and there are RS232 connections to the on board processors for bootstrapping, debugging, configuring, etc.

A. Protocol Offload Section

The protocol offload section of the ONN is designed to allow much of the communication protocol work to be done in it, instead of the attached host computer. Our design strategy was to try to reduce software CPU instruction path-length counts as much as possible. This is because there are very few instruction cycles available to process each packet, and

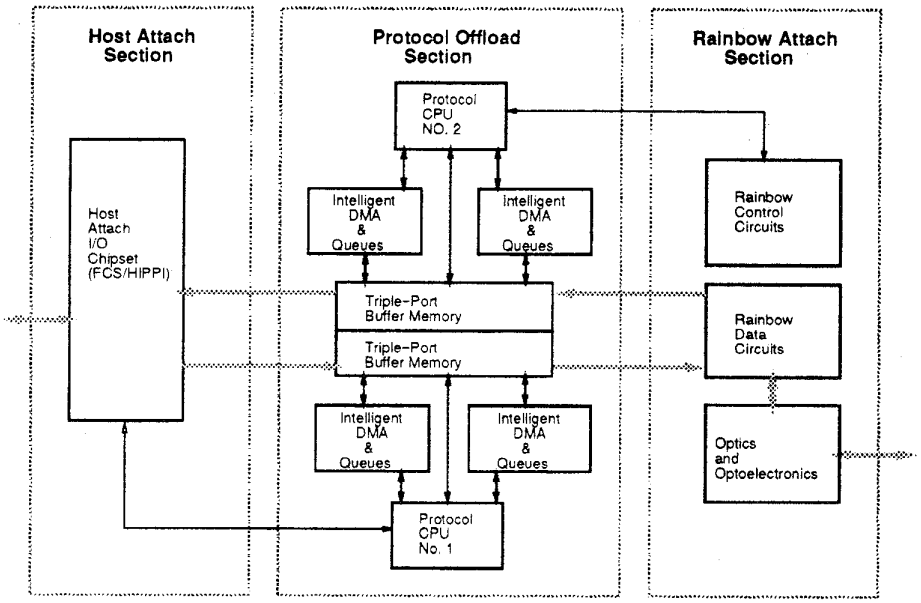

Fig. 2. The optical network node.

so the software overhead is typically the bottleneck for high bandwidth and low latency. (For example, in order to achieve 1 Gbit/s throughput using 4 KB packets, each packet must be processed in 32 μs; with a 10 MIPS processor this means that all processing for a packet has to be done in less than 320 instructions.) This strategy has led to a number of design decisions as follows.

- The operating system overhead in many protocol implementations is large, and in many cases cannot be reduced. In our system this was considered a key area, and so an operating system, per se, is not used. The code is a dedicated program running directly on the hardware, with a number of support functions such as memory management, etc. being fine tuned to the hardware and the application. This does lead to increased programming effort, both to code the support infrastructure, and because an existing (e.g., UNIX-based) implementation of the transport protocol cannot not be easily ported to this environment.
- Given that the hardware is being implemented specifically to assist in the protocol offload, a number of hardware features were designed into the system to improve the performance of the ONN by assisting the onboard software. The protocol offload section is comprised of two individual protocol processor boards, one for each of the two directions of data travel. The performance enhancing features on each board include a protocol microprocessor, microprocessor program memory, data/header buffer memory, two intelligent direct-memory access (DMA) controllers, four DMA buffer queue FIFO's, inter-CPU communication FIFO, an inter-CPU DMA, and hardware calculation and checking of checksums.

1) Protocol CPU: Each protocol processor board contains a high-performance RISC CPU (Intel i80960CA) which is responsible for handling buffer management and protocol offload for data traveling in a given direction. The i80960CA processor is a 32-bit RISC CPU which has been used elsewhere in similar applications.

2) CPU Program Memory: Each protocol processor has 2 MB of separate RAM memory to contain the ONN software and its associated variables and data structures. The program memory is separate from the data memory to improve access speed. The ONN control software is loaded into this memory via the RS232 ports from an attached system such as a PC.

3) Triple Port Data/Header Buffer Memory: In order to minimize contention for the memories, it was decided to try using two separate memories on separate buses, one for data traveling from the network to the host computer, and one for data traveling in the other direction, in the hope that this will allow data to travel in both directions concurrently, while eliminating serious contention for the memory busses.

Triple-ported video RAM's are used for this memory. The video RAM's have two "sequential" interfaces which are used for the DMA access to and from the memory and a random-access interface which is used for the interface to the CPU.

4) Intelligent DMA Controllers: Each protocol processor board has two DMA controllers for data traffic, one to handle data movement into the buffer memory, and the other to handle movement out of the buffer memory.

These DMA controllers are "intelligent" in the sense that they are capable of using the buffer queue FIFO's (described below) to obtain and free buffers in memory. They are therefore asynchronous (to some extent) from the operation of the protocol processors, freeing these processors from the cycle-consuming task of setting up each DMA operation and processing DMA complete functions. The "intelligent" features of these DMA controllers help eliminate some software overhead, thus improving overall ONN performance.

In addition, to eliminate the necessity of copying or moving data in memory, the protocol headers are stripped by the DMA

controllers from the user data portion of the packets, and are placed in separate memory buffers, dedicated to header use. This allows different headers to be prepended to the data on the outgoing side, without the necessity of moving data in memory, again assisting performance.

To support this header/data stripping and to assist buffer management, the DMA controllers operate with two sizes of buffers, called H buffers and D buffers. The H buffers are intended to hold protocol headers and control block information, and are therefore relatively short, and the D buffers are intended to hold user data, and are relatively long. The sizes of the buffers are selectable by software, and the H/D distinction is managed by the DMA controller hardware as well as the software.

The transmit DMA controller, in order to allow for building up of transmission units from smaller memory blocks, supports appending of DMA entries via an *append* function, and also allows such units to fall on arbitrarily aligned addresses, and have arbitrary lengths (no restrictions as to 32-bit boundaries or 32-bit multiples). This allows protocol software to build up packets from various pieces in disparate portions of memory, allowing for different protocols, protocol conversion, etc. without having to move data in memory. This again enhances performance.

The receive DMA controller has support for calculation of a TCP checksum on the received packet. When (and if) TCP/IP is implemented in the node, this will provide a very fast method of calculating the necessary checksums, which is considered a serious performance bottleneck in most systems that do this calculation in software.

5) DMA Buffer Queue FIFO's: These 32-bit wide queue FIFO's are intended as an interface between the protocol processors and the "intelligent" DMA controllers, and also are a simple mechanism to assist the processors in performing buffer management. The FIFO's queue up requests for transmission, free buffers for reception, and provide completed transmission and reception status information. These queues eliminate the necessity of immediate software response to events, since no event is lost if the software cannot respond within a small time interval. This eliminates interrupt overhead, thus increasing performance.

There are five logical DMA buffer queue FIFO's on each CPU board:

- A *receive complete* queue: This queue FIFO contains the addresses, lengths, buffer types, and error status of buffers that contain received data packets, received headers, and received "control" blocks. Each entry in this queue FIFO consists of the following "fields":
  a) the address of the buffer in buffer memory;
  b) the length of the received information in memory;
  c) the type of block received (e.g., header, data, connection, disconnection);
  d) the error status;
  e) the type of buffer (i.e., H buffer verus D buffer); and
  f) the 16-bit one's complement TCP/IP checksum.
- Two *free receive buffer* queues: These queue FIFO's contain the addresses of H and D buffers to be used by the "intelligent" DMA for subsequent receive operations. There are two free buffer queue FIFO's, one for H buffers (for headers) and one for D buffers (for data). The "intelligent" DMA removes an entry from a free buffer queue FIFO at the beginning of each receive operation when it needs to determine where in buffer memory to place the data or the header.
- A *transmit data* queue: This queue FIFO contains the addresses and lengths of buffers that contain data packets and headers to be transmitted, as well as special "control" buffers to be sent to the I/O adapters for other functions (e.g., connection set up, etc.). Each entry in this queue FIFO consists of the following "fields":
  a) the address of the buffer in buffer memory;
  b) the length of information to be transmitted via the DMA;
  c) the function code for the buffer (e.g., header, data, connect, disconnect, etc.); and
  d) a tag field used by the software to keep track of the data block—this tag is returned in the transmission complete queue below.

  The appropriate protocol processor builds these entries and places them in the queue FIFO when it has data or headers that need to be transmitted, or when it has special commands (e.g., connect) that it wishes to send to the I/O adapter.
- A *transmission complete* queue: This queue FIFO contains the addresses of header, data, and control buffers just transmitted by the "intelligent" DMA. Each entry consists of the following "fields":
  a) the address of the transmitted buffer in memory,
  b) the function code (from the work queue entry) of the buffer,
  c) the error status (e.g., OK, parity error, connection rejected, etc.), and
  d) the software tag field set in the transmit data queue above.

6) Inter-Processor Communication Message Queue FIFO's: These message queue FIFO's are similar to the buffer queue FIFO's. However, they are used for communication between the two protocol processors. There is one message queue FIFO for each direction of communication. Once again, these queues enhance performance by queueing up inter-processor communication information so that immediate software response is not required to prevent information loss. Fast inter-CPU communication is a requirement for most communication protocols, since the receiver and transmitter must be able to communication for purposes of acknowlegment of received packets, for example.

Each FIFO is 32-bit wide, and the information transmitted is determined by the protocol software. This information consists of software "messages" to be passed from one processor to the other.

7) Inter-Processor DMA: The inter-processor DMA is used when it is necessary to transfer large amounts of information from the data memory (VRAM) of one protocol processor card to the other. At this point in time, this feature is not used for

protocol processing but is only used to download code from one processor to the other or for diagnostic/control purposes.

Each protocol processor card has a DMA controller that can be used for three purposes:

- transferring data from this CPU VRAM to the other CPU VRAM (called *remote copy*);
- transferring data from one location in the local VRAM to another location in local VRAM (called *local copy*); and
- calculating a checksum on an area of local VRAM without copying (called *checksum only*).

In addition, when the remote copy function is used, the DMA controller has a special function of copying an eight word (32 byte) "message" from a special fixed location in local VRAM to another special fixed location in the remote VRAM, at the completion of the remote copy function. This message is used by the software to communicate command and control information associated with the block of data that was transferred via the DMA.

B. Optical Network Attach Section

The network attach section of the ONN consists of two cards, an optical/analog card, and a digital card.

1) Optical/Analog Card: The optical/analog card contains a distributed-feedback (DFB) laser, a fiber-Fabry–Perot tunable filter, and an avalanche photodiode (APD) receiver. Each ONN in the network has a laser at a different wavelength and the laser is maintained at a constant temperature to avoid wavelength drift. The Fabry–Perot filter has a free spectral range of 32 nm and a finesse of about 100, and can tune from one wavelength to another in 25 ms.

2) Digital Card: The digital card implements the *circular search* medium access control protocol used in Rainbow-I [4], and provides flow control in hardware.

3) Circular Search Protocol: The Rainbow-I network used a simple inband signaling protocol to set up and takedown connections between nodes in the network. Essentially, in order to set up a connection, a node continuously broadcasts a connection request message on its assigned wavelength. The intended destination, if idle, scans its tunable filter across all the wavelengths looking for such a request and locks to a wavelength if it sees such a request. It then sends back a connection accept message that the originator looks for while itself scanning across all the wavelengths. In Rainbow-I this protocol was implemented mostly in software. Rainbow-II implements this protocol entirely in hardware, mainly to relieve the protocol processing CPU's from performing this function.

4) Hardware Flow Control: Once a connection is set up between two nodes, flow control is required to prevent packets from being lost due to buffer overflows at the receiving node if it is unable to cope with the speed of the sender. A *sliding window* mechanism [13, chap. 2] is commonly used for this purpose. At each instant the sender maintains the value of the maximum sequence number that he can transmit and the last acknowledged sequence number. Based on its available buffers, the receiver sends tokens periodically to increase the value of the maximum sequence number that the sender is allowed to transmit. Usually such a scheme is implemented in software. Rainbow-II implements this entirely in hardware on the digital card, again to partially relieve the protocol processing CPU's from performing this function.

III. Software Design

The ONN software supports two modes of operation.

A) *Transparent* mode: In this mode, once a connection is established [using the HIPPI LE (link encapsulation) header to determine the destination], packets are passed through the ONN without modification. The protocol offload function is disabled in this mode. This mode allows existing TCP/IP or other protocols on the host to operate across the network end-to-end without any changes.

B) *Protocol offload* mode: This mode allows the complexity of a full reliable transport protocol to be offloaded from the host computer to the ONN. The protocol stack used for this purpose is shown in Fig. 3. The host computer communicates with the ONN via a simple, lightweight protocol called simple host intersocket protocol (SHIP) [14] that implements the semantics of the Unix BSD sockets interface. A socket-like library is provided to host applications. Thus, host applications only have to be relinked (not rewritten or recompiled) to the new library in order to benefit from the performance of protocol offload. The ONN implements the reliable transport layer protocol internally to talk to the peer ONN. Currently, the transport protocol implemented between Rainbow II nodes is a simple, light-weight, easy-to-implement private protocol called optical transport protocol (OTP) which was used initially to save effort. Later, this may be replaced with a full TCP/IP implementation for two purposes: to connect with hosts and other networks running TCP/IP, and to compare the performance of a TCP implementation with the OTP implementation.

The main goal of the software was to eliminate as much software path-length as possible. This was done via the following design decisions.

1) No operating system or kernel is used inside an ONN. The ONN software runs completely stand-alone, specifically tailored for the ONN hardware.
2) No multitasking is used. Although this would have eased the construction of the software, it would have used precious CPU cycles. So the software runs in a "polling loop" implementation, where each function runs sequentially and cannot wait for an event or resource.
3) There are no interrupts under normal operation. This also reduces CPU cycle utililization and was made possible by the widespread use of FIFO's in the hardware design, thus eliminating the need for immediate response to events by the queueing function of the FIFO's.
4) In order to support different transport protocol software (currently OTP, future TCP/IP), the interface between SHIP and the transport protocol was kept as generic

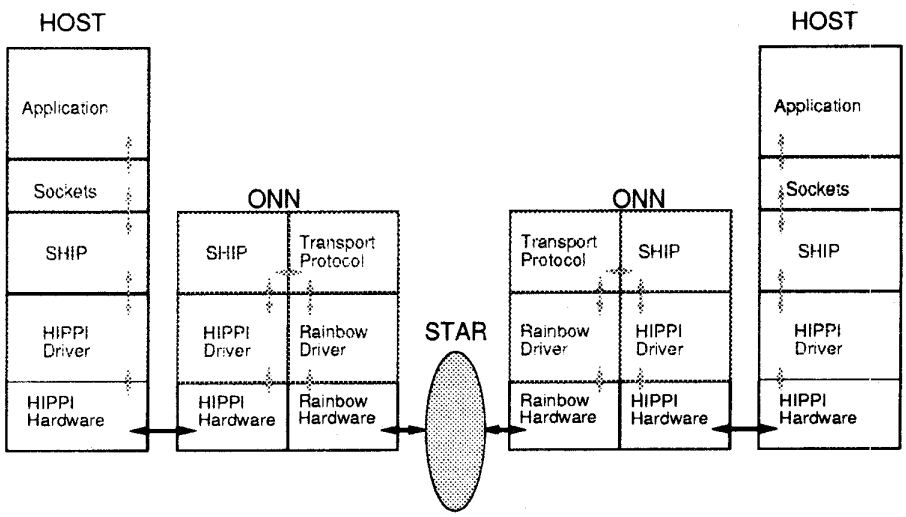

Fig. 3. The protocol stack.

as possible. Subroutine calls for opening and closing connections, transmitting packets, and handling errors are the only interfaces between the two protocols. There is no sharing of control blocks or other memory-based structures.

A. SHIP

The protocol offload hardware support in the ONN surfaces another problem, namely how to get the data between the computer running the application program (referred to below as the host) and the protocol offload hardware in the ONN. To solve this problem, we developed the SHIP protocol [14] jointly with Intel. SHIP has been implemented on a variety of hosts (Cray YMP, Intel Delta, and IBM RS6000) and two protocol processors (PP's)—the ONN and the crossbar interface (CBI) at LANL [15].

This immediately brings up the question "Why is SHIP less work for the host than the protocol being offloaded?" There are a number of answers.

- The connection between the host and the PP is a point-to-point dedicated connection, or a circuit-switched connection, not a packet-switched network. It may, in some future implementation, even be a memory-to-memory or internal bus connection to a PP that is part of the same computer as the host. Thus, there is no need in SHIP for routing, congestion control, packet fragmentation and reassembly and other protocol functions used in full network protocols, such as TCP/IP, or other proposed "lightweight" protocols that have attempted to solve a similar, but not identical problem.
- The connection between the host and the PP is assumed to be much more reliable, in terms of corrupted bits, than a "normal" network connection, allowing error recovery to be simpler, since the number of retransmissions is assumed to be very low and therefore not an area needing optimization. This, in conjunction with the point-to-point nature of the connection, allows for very large packet sizes, reducing the per-packet protocol overhead considerably.
- Since SHIP, by definition, will only be seen between the host and the PP, not on any external network, there is no need to support backward compatibility with a large zoo of existing protocols. This allows a number of simplifying design decisions to be made.

1) SHIP Design Assumptions: The SHIP protocol was designed with two important objectives: 1) It must allow existing application programs to run with as few changes as possible, preferably none. 2) It must be much less work for the host to process than the protocols being offloaded (TCP/IP).

The first objective led to the design decision that the SHIP protocol would basically be a protocol implementation of the sockets interface, which is widely used as the application interface to TCP. Ideally, applications that use sockets to access TCP would need only be relinked, or in some cases only need to specify different run-time parameters, and would then be using SHIP instead of TCP to talk to a PP that would then do the protocol processing. Therefore, SHIP is a protocol that essentially imitates the sockets interface by an actual protocol transmitted over a point-to-point connection.

The second objective led to a number of important design decisions.

- All SHIP headers are essentially of one common fixed length and fixed format, except for a few header types containing variable length extensions at the end.
- Most fields in the SHIP headers are 32-bit aligned fields.
- Data packets can be very large, potentially gigabytes in size. The actual size used is negotiated between the host and the PP on a per-transfer basis (i.e., not fixed per connection). Currently, due to buffer memory limitiations in the ONN, packets of 1 MB are considered typical.
- The protocol is a master–slave protocol, not peer-to-peer. The host is always the master, and the PP is always the slave. This has the slight disadvantage that SHIP cannot not be used as a lightweight transport replacement for

TCP between hosts, but requires an intermediate PP. The reduction in overhead in both the host and PP more than offsets this disadvantage.

- A number of simplifying restrictions have been made, such as the following.
  a) A host cannot send or request data out of order, except for repeating a transmission not yet acknowledged.
  b) Data, once received by the PP and acknowledged to the host has not necessarily been received by the remote peer. It is the PP's responsibility to ensure reliable delivery.
  c) Certain features of TCP, e.g., "urgent data" are not supported, for performance reasons.

2) SHIP Overview: The standard socket calls that are supported by SHIP are: *socket, bind, connect, listen, accept, shutdown, select, getsockopt, setsockopt, read, write, send/sendto, recv/recvfrom,* and *close*. The blocking and nonblocking versions of the appropriate commands are also provided.

The SHIP packet format itself consists of a fixed format header, consisting of 10 32-bit words, optionally followed in some cases by a variable-length extension, which must be an even integral number of 32-bit words (in order to support some link implementations that are 64 bit in width, such as 1600 Mbit/s HIPPI).

The fixed fields in the SHIP header are the following.

Function Code: This is a 16-bit field (in the first 32-bit word) that indicates the specific function (e.g., create, read, etc.) that the packet is calling for.

Host Id: This is an 8-bit field (also in the first 32-bit word) that is used when a PP is attached to multiple hosts.

Status: This is an 8-bit field (also in the first 32-bit word) that is used by the PP to return any protocol errors to the host.

Request Id: This is a 32-bit word that is a unique number specified by the host for each Requested function, and returned by the PP with each reply, allowing the host and the PP to detect retransmissions.

Host Socket ID: This is a 64-bit number used by the host to match the replies from the PP to its own internal socket identifiers. It is sent by the host in each packet, and is returned by the PP in each reply. The PP has no other use for the field, and never interprets or modifies it.

PP Socket ID: This is a 64-bit number, used by the PP to match the requests from the host to the PP's own internal socket identifiers. It is initially returned to the host when a socket is created (via the create function) and is then sent by the host in each request packet for that socket. The host neither interprets nor modifies this field.

Note: The use of the above two fields to identify sockets allows the host and the PP implementations to use their own optimized identifiers for sockets, allowing improved performance in the implementations.

Extension Length: This is a 32-bit word that contains the length of the optional variable length extension to the SHIP header. In most SHIP packets it is zero. Only a small number of SHIP packets require more information than can fit in the fixed portion of the header, and these use this field.

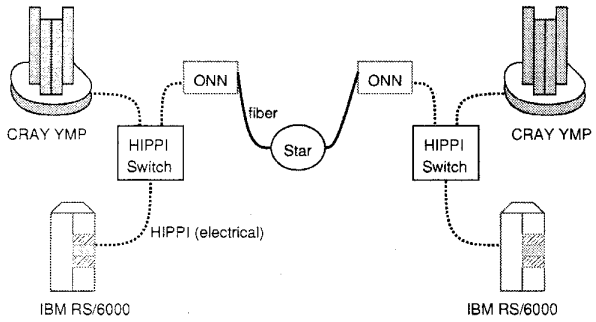

Fig. 4. The LANL testbed.

Data Length: This is a 32-bit word that contains the length, in bytes, of the Data portion of the packet, which immediately follows the header (for data transfer packets only, such as read or write).

Command Status: This is a 32-bit word that is used by the PP to return any errors occurring in the execution of the specific socket command being issued (e.g., read, accept) as opposed to SHIP protocol errors which are returned in the Status field.

Parameter Word: This is a 32-bit word that is used for various parameters that differ for different SHIP functions. Since a number of functions needed at most one parameter, it was decided to have this word in all headers, rather than to force many headers to use the variable-length extensions just for a single word. An example of the field's use is for the READ command SHIP header, in which this field contains the size of data being requested.

As can be seen, the SHIP headers were designed to make the header processing as simple as possible for the host and the PP, without too much concern for the possible extra transmission overhead that might be associated with 32-bit fields and fixed-length headers. Since the whole purpose of SHIP was to allow operation on very-high speed networks, this small transmission overhead is insignificant.

B. Optical Transport Protocol

OTP is a very simple protocol used as a transport layer protocol between Rainbow II nodes. This protocol was used because it was very easy to implement and has a potential for high performance. Standard transport layer protocols such as TCP were not initially needed, since Rainbow II nodes can only talk to other Rainbow II nodes, so the transport Layer protocol used on the Rainbow II network is never surfaced to any other entity.

The OTP protocol is a go-back-N protocol using fixed length packet headers and is capable of using large packet sizes.

The go-back-N error recovery strategy is not the most efficient, but it is easy to implement, and works well because the optical links have a very low error rate and packets are not dropped due to buffer overflows because of the hardware flow control feature.

A fixed length packet header with fixed length 32-bit fields is used. This allows for fast simple access to packet header information. In fact, to simplify the implementation, the OTP

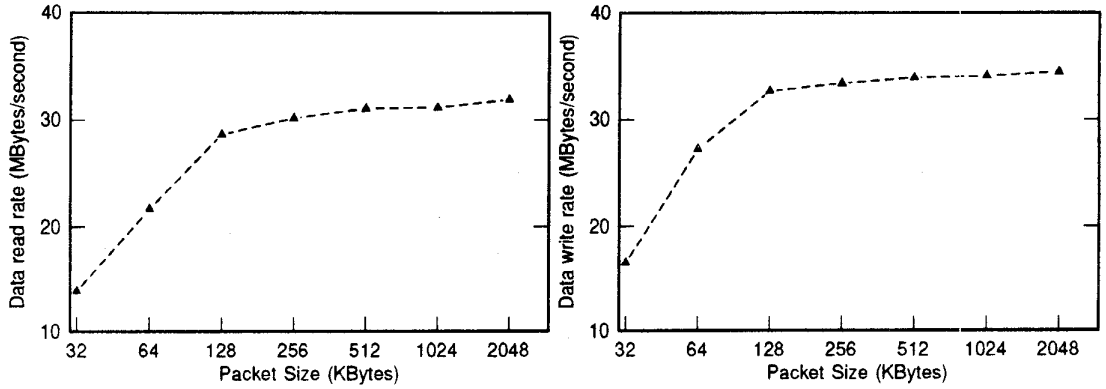

Fig. 5. Raw data transfer rates between two RS6000 workstations across the Rainbow-II network. The ONN's are operated in transparent mode.

packet header is identical in size, and layout to the SHIP packet header, with a few fields used for different purposes.

OTP allows packets to be very large (megabytes), allowing for larger packet processing times. This is a potential disadvantage if the error rate is high, since retransmissions use up a lot of bandwidth, and the probability of a packet being received in error is higher. However as the Rainbow II error rate is low, this was judged not to be a problem. The implementation of OTP uses this packet size for another advantage, namely the incoming SHIP packets are converted directly to OTP packets by merely changing a few fields in the header, with no packet disassembly or reassembly required.

IV. Status and Experimental Results

The Rainbow-II network was demonstrated at Supercomputing'94 and at the Optical Fiber Communication Conference'95. The demonstration had three RS6000 workstations, each connected to an ONN. All the ONN's operated in transparent mode. Two nodes served as video servers, sending out high resolution images. The third node served as a receiver and could establish connections to each of the other nodes across the network. Each image was 5 MB and about 12 images were transmitted per second across the network on a connection as 1 MB packets. The resulting 60 Mbit/s throughput was limited primarily by the speed at which we could get data out of the RS6000's and not by the ONN's.

Rainbow II is currently deployed at LANL. LANL is evaluating the performance of the Rainbow-II network against established gigabit testbeds, such as the LANL HIPPI Testbed, by comparing data-transfer rates between supercomputers/workstations and other high-end HIPPI-based systems. The LANL testbed network in its current form is shown in Fig. 4. It consists of two ONN's interconnected by a passive star coupler (with a third yet to be connected). The ONN's are not directly connected to host machines. They are instead connected through the existing HIPPI network to either IBM RS6000 workstations or to Cray YMP machines.

Figure 5 shows the test results for sending raw data through the ONN's in transparent mode. The data was sent from an RS6000 through the ONN's and received back by the same RS6000. The data is taken from the average of five runs with each run consisting on sending 100 packets between the two machines. The *read* rate is the rate at which a host receives data from an ONN and the *write* rate is the rate at which a host can send data to an ONN. As expected, the data rates increase with the packet size since there are fewer packets to be processed per unit time, and for large packets we are able to obtain about 30-35 Mbit/s data rates on average. This is consistent with the 60 Mbit/s rates we were able to get earlier between two RS6000's, and again is limited by the RS6000's and not by the ONN's.

Figure 6 shows the test results for data transfer between two Cray YMP's. In addition to the data read and write rates we also plot two additional parameters of interest: a) *CPU utilization*, which is defined as the ratio of the CPU time taken to transfer the data to the total time taken, and b) the *data transfer efficiency*, which is the the data rate divided by the CPU utilization, or in other words the total amount of data transferred divided by the CPU time taken. In each plot, one curve is from an experiment wherein the Crays are connected through the ONN's with the ONN's operated in protocol-offload mode. The Crays and the ONN's run SHIP in this case. The other curve corresponds to an experiment wherein the two Crays are connected to each other through the HIPPI network without going through the ONN's. The Crays run TCP in this case. In each case, the data is averaged from several runs.

Observe that the SHIP write rate is higher than the read rate. This is probably because the two Crays were operating under different loads at the time of the tests. We are verifying this further currently. The maximum obtained SHIP write rate during a run was 78 Mbit/s which is close to the 100 MB/s maximum achievable rate across a HIPPI connection, given that there is some added overhead for framing and interpacket delays.

Observe that the average TCP transfer rate of around 55 MB/s is almost independent of the packet size. This is because the maximum internal packet size of TCP is 64 KB and larger packets are segmented and actually sent in 64 KB blocks. Note that for large packet sizes, SHIP achieves higher transfer rates and lower CPU utilizations, or equivalently, higher transfer efficiencies, than TCP. This illustrates the benefit of protocol offload. There are three reasons for this: a) SHIP uses larger-sized packets (up to 1 MB in the current implementation) without segmenting them into smaller packets, b) SHIP has

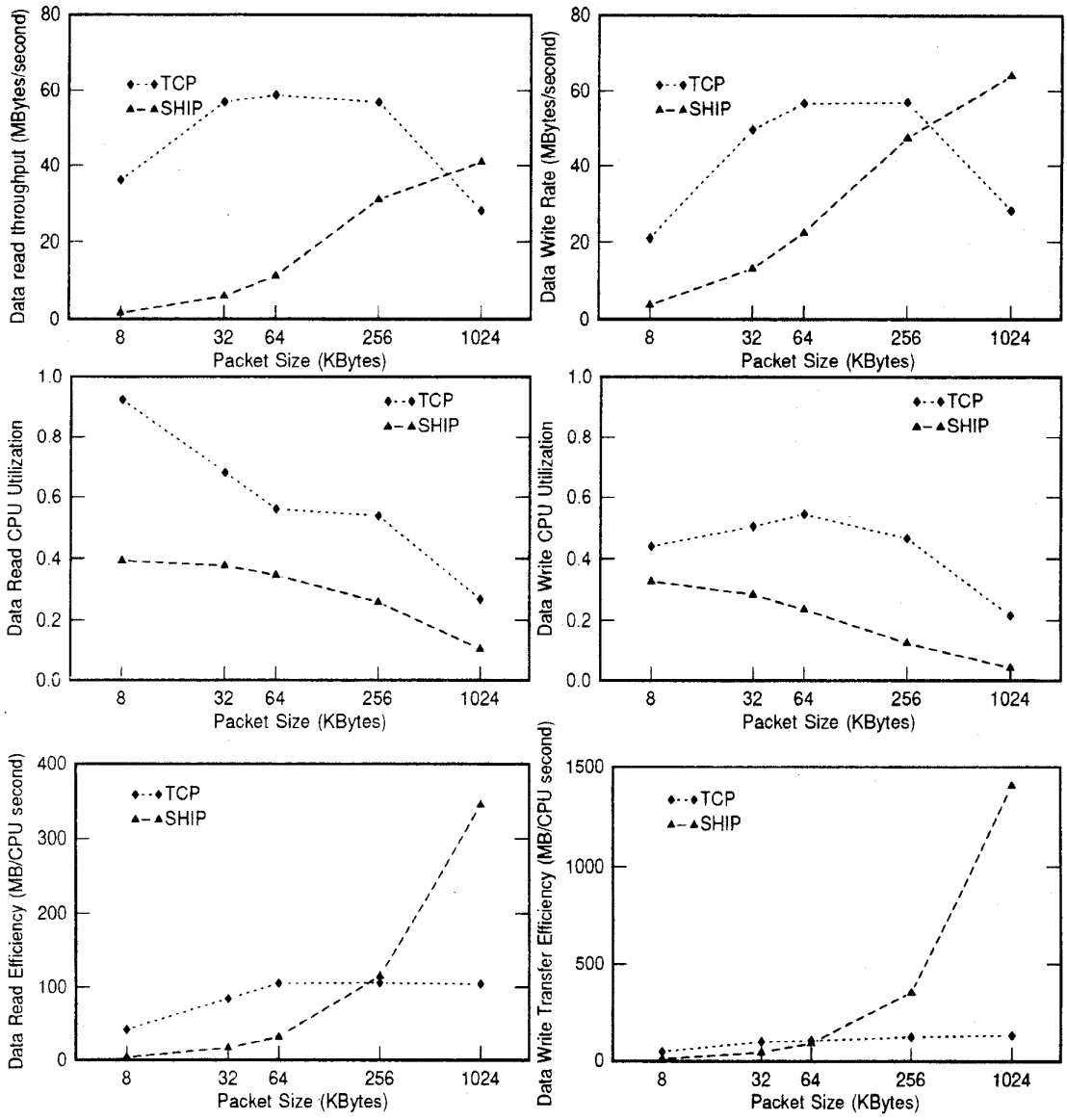

Fig. 6. Data transfer rates, CPU utilization and data transfer efficiency between two Cray YMP machines. The curve labeled "SHIP" corresponds to the machines connected through two ONN's in protocol-offload mode, running SHIP. The curve labeled "TCP" corresponds to the machines connected to each other directly through the HIPPI network, running TCP.

no window-size limitation, while the TCP window size on the Crays is 500 KB, and c) SHIP is inherently designed to be a more efficient protocol than TCP. Given c), it is surprising that the SHIP transfer rate is lower than the TCP transfer rate for small packet sizes. One reason for this may be that TCP is implemented very efficiently in the kernel of the Cray, while SHIP is currently an application program that has not yet been tuned to obtain the best performance. By putting SHIP in the kernel, the host could process (add SHIP headers) and send SHIP packets without being interrupted and swapped as in user mode, and the processing of the packet in the kernel could be done on another processor of the Cray while the application continues processing the next packet. Another reason is that the current implementation of SHIP is essentially as a stop-and-wait protocol. Allowing queueing of messages so that the host could send them back-to-back would help to boost performance. As currently implemented, the host will not send another packet until the current one has been acknowledged. By allowing back-to-back packets, there would be less gaps between packets. We are exploring these further.

V. Future Work

LANL plans to run actual applications across the ONN's using both transparent and protocol-offload modes. The transparent mode is more useful for some applications that do not use the Crays (for instance global climate modeling) and that transfer non-IP (internet protocol) data between the Thinking Machines CM-5 and a high performance data system (HPDS) which is based on a large IBM Disk Array. Another one is a

virtual reality (VR) application running between the Crays and an SGI workstation. This application will transfer data from the Cray at 800 Mbit/s and use the SGI to render the images for subsequent display. The high speed link is used to speed up the overall VR processing. LANL is currently interested in exploring technologies for 10 Gbit/s MAN's to support some evolving applications and networks like Rainbow-II offer the potential to be scaled up to achieve these rates.

The ONN has been designed to support a variety of host interfaces as well as to support optical packet switching in future as rapidly tunable filters become available. Among the five cards in an ONN, only the network attach analog and digital cards need to be changed. The ONN software and the rest of the hardware would be unchanged. We are currently developing components and protocols that will enable us to support optical packet switching.

REFERENCES

[1] N. K. Cheung, G. Nosu, and G. Winzer, Eds., "Special issue on dense WDM networks," *IEEE J. Select. Areas Commun.,* vol. 8, Aug. 1990.

[2] M. J. Karol, C. Lin, G. Hill, and K. Nosu, Eds., "Special issue on broadband optical networks," *J. Lightwave Technol.,* May/June 1993.

[3] P. E. Green, *Fiber-Optic Networks.* Englewood Cliffs, NJ: Prentice-Hall, 1992.

[4] F. J. Janniello, R. Ramaswami, and D. G. Steinberg, "A prototype circuit-switched multi-wavelength optical metropolitan-area network," in *Proc. ICC'92,* 1992, pp. 818–823.

[5] "High-performance parallel interface," American National Standards Institute, X3T9.3, rev. 6.9, Nov. 1989.

[6] "Fiber channel: Signalling protocol (FC-2)," American National Standards Institute, X3T9.3, rev. 4.2, Nov. 1994.

[7] W. Doeringer, D. Dykeman, M. Kaiserswerth, B. Meister, H. Rudin, and R. Williamson, "A survey of light-weight transport protocls for high-speed networks," *IEEE Trans. Commun.,* vol. 38, pp. 2025–2039, Nov. 1990.

[8] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An analysis of TCP processing over-head," *IEEE Commun.,* vol. 27, no. 6, pp. 23–29, 1989.

[9] H. Kanakia and D. R. Cheriton, "The VMTP network adaptor board (NAB): High-performance network communication for multiprocessors," in *Proc. ACM SIGCOMM'88,* 1988, pp. 175–187.

[10] E. C. Cooper, P. A. Steenkiste, R. D. Sansom, and B. D. Zill, "Protocol implementation on the Nectar communication processor," in *Proc. ACM SIGCOMM'90,* 1990, pp. 135–144.

[11] M. Kaiserswerth, "The parallel protocol engine," *IEEE/ACM Trans. Networking,* vol. 1, pp. 650–663, Dec. 1993.

[12] D. C. Feldmeier, "An overview of the TP transport protocol project," in *High-Performance Networks—Technology and Protocols,* A. Tantawy, Ed. Norwell, MA: Kluwer, 1993.

[13] D. Bertsekas and R. G. Gallager, *Data Networks.* Englewood Cliffs, NJ: Prentice-Hall, 1992.

[14] B. Beattie, M. G. Halvorson, C. Idler, J. Kravitz, M. Sukalski, and R. Thomsen, "SHIP—simple host intersocket protocol," working document, Feb. 1995.

[15] C. Idler and R. Thomsen, "The LANL crossbar interface: Functions and performance," in *Proc. Networld Interop,* Mar. 1995.

Eric Hall received the B.S.E.E. degree with highest honors from Drexel University, Philadelphia, PA, in 1975.

He is a Senior Engineer in the Optical Network Systems group at the IBM T. J. Watson Research Center, Hawthorne, NY, where he designs and implements high-speed optical systems. He also worked at Burroughs, Floating Point Systems, and Multiflow Inc., in the area of supercomputer design and high-speed interconnects before joining IBM in 1990. He is on the Program Committee of the Optical Fiber Communications Conference, 1997.

Jeff Kravitz, photograph and biography not available at the time of publication.

Rajiv Ramaswami (S'88–M'90–SM'96), for a photograph and biography, see this issue, p. 762.

Marty Halvorson is a software engineer working at the Los Alamos National Laboratory, Los Alamos, NM, on very high-speed network research projects. Prior to his current efforts, he was at Digital Equipment Corporation, where he also worked on very high-speed networks. He also represented Digital Equipment at the ANSI Standards group that created the HIPPI standard, and was Chairman of the industry group that created the Serial-HIPPI technical specification, which is currently in the process of becoming a standard.

Stephen Tenbrink received the undergraduate degree from the University of Denver, CO, in 1970, and the M.S. degree in electrical engineering from the University of California, Los Angeles, in 1977.

He is a Technical Staff Member in the Network Engineering Group of Los Alamos National Laboratory's (LANL) Computing, Information, and Communications (CIC) Division. Since 1980, his work has focused on the development of high-performance computer networks—initially with a LANL developed 50 Mbit/s interface and evolving to Gbit/s interfaces. He was the section leader for the group of engineers that developed the HIPPI-based systems at the Laboratory. His current work involves research and development in new networking technologies such as ATM. He is working with other Laboratory engineers to build an ATM testbed for the Laboratory and in the deployment of gigabit network technology (HIPPI and ATM) into the Laboratory's production computing facility.

Richard Thomsen is a Staff Member working as a software engineer in the Computer Networking group at Los Alamos National Laboratory, Los Alamos, NM. He is involved in developing the software for the high-speed networking implementation, and was involved with the ANSI meetings when the HIPPI standard was developed.